Report

Question 1

Other solutions:
The problem involves calculating the sum of distances between every pair of black nodes in a given tree. The tree is represented by its nodes, each having a color (black or white), and edges with corresponding weights denoting their lengths.
A straightforward way to solve this is to select a node as the root and iterate through all nodes in the tree using dfs and count the sum of distances from this root to every black node. Then let every nodes to be the root, and use an array to store the total distances so we need to iterate n times. Last we multiply the color index with the distance index (we only need the distances of black nodes to other nodes) and sum the array then divide it by 2 (each pair of black nodes calculated twice). This way, the time complexity would be O(n^2).

My solution:
I realize the tree structure and a dfs to iterate through would be a must. In reducing the time complexity based on this, I saw a problem requires to return an array storing the distance of each nodes to all other nodes. The idea is information of each nodes (like subtree's number of nodes) can be accessed and stored during the first dfs, and then every node's distance to all other nodes can be calculated based on these information without doing dfs over again. This idea can be applied to this problem.
I first used a dfs with the first nodes to be the root. In this process, store the information of each node's 1) number of black nodes in its subtree in the array 'count' 2) total distance to all black nodes in its subtree in the array 'dis'. Set an array 'ans' to store the summed distance and we can store the first node u's dis in it now. Then in function 'dfs2' we use equation ans[v] = ans[u] - count[v] * w + (black - count[v]) * w (the equation is easy to get if we draw a graph) to calculate to calculate u's child node v's total distance. Then get all answers using a recursion. With the ans array the next step is like what's in the other solution, multiply the numbers by the color since we only want the black ones and then divide it by two since each pair is calculated twice.

My solution has a time complexity of O(n), because only in the first dfs we iterate through the tree. Then for each node we only need 1 operation (instead of n) to get the answer based on info provided by the first dfs, which is much more efficient.

Question 2

Other solutions:
The problem involves managing a sequence of prices through three types of operations: buying a new book, finding the minimum absolute difference between adjacent prices, and finding the absolute difference between the two closest prices in the entire sequence. The operations are performed on a sequence of prices. There should be two functions, closest_adj_price () and closest _price (). Also, we need an operation () to handle buying a new book and find the new closest adjacent prices and closest price after buying the book.
To find closest_adj_price(), the method can only be iterate through the array and calculate the

difference, and the time complexity is O(n). To find the closest price, the idea would be first sort the numbers, and then calculate the difference of each two adjacent numbers in the sorted numbers, and return the minimum difference. This can have many solutions as there are a lot of sorting algorithms. To implement operation (), an easy to think way is just insert the new value and call the two functions again to calculate the new closest and closest_adj prices.

My solution:
Calling closest_adj_price() and closest _price() every time we buy a new book would waste much time. We can run both functions once and get the initial two values (closest_adj_price and closest _price), then for each newly bought book, update these values by limited steps of operations. Specifically, for closest_adj_price(), we only need to calculate difference between the inserted number and the last number in the array, and compare it with the previously calculated closest_adj_price to see which is smaller. And for closest _price (), we only need to insert the new number in the sorted numbers and calculate the difference between it with its former and later number, and compare them with the previously calculated closest _price.
I used a binary search tree for the sorting in the closest _price function, methods that are needed in this problem like insert, finding predecessor, finding successor are basic operations in BST and all have a time complexity of O(logN), which is more efficient than data structures like array or linked list. The first sorting has a time complexity of O(nlog(n)).

My solution improved efficiency by simplify the operation after buying a new book and using BST in finding closest price.